VerChor: A Framework for the Design and Verification of Choreographies

Matthias Güdemann, Pascal Poizat, Gwen Salaün, Lina Ye

VerChor: A Framework for the Design and Verification of Choreographies

Matthias Güdemann, Pascal Poizat, Gwen Salaün, and Lina Ye

***Abstract*—Choreographies are contracts specifying from a global point of view the legal interactions that must take place among a set of services. Such a contract may serve as a reference in the development of concurrent distributed system, whether it is achieved following a top-down or a bottom-up approach. In this article, we present VerChor, a generic, modular, and extensible framework for supporting the development based on choreographies. It relies on a choreography intermediate format (CIF) into which several existing choreography description languages can be transformed. VerChor builds around a set of formal properties whose verification is central to choreography-based development. To support this development process, we propose a connection between CIF and the CADP verification toolbox, which enables the full automation of the aforementioned properties. Finally, we illustrate a practical use of the VerChor framework through its integration with the Eclipse BPMN 2.0 designer.**

I. Introduction

APPLICATIONS are now often constructed out of the reuse and assembly of distributed and collaborating peers, *e.g.*, software components, Web services, or Software as a Service in cloud environments. In order to facilitate the integration of these independently developed components, that may reside in different organizations, the peers participating in a composition should adhere to a global contract. Such a contract, called *choreography*, specifies from a global point of view the interactions that must take place among a set of peers. It is a reference for the further development steps, *e.g.*, service selection, discovery, composition generation and evolution.

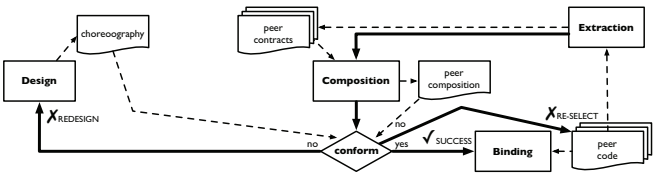


Fig. 1. Bottom-Up Development Process

Choreographies support *bottom-up* development (Fig. 1). Several peers have been selected to be composed. They may exhibit behavioral contracts or behavioral contracts can be retrieved from them [1]. Here, one has to check that the composition of the peers has exactly the same behavior than what was prescribed in the requirements, *i.e.*, the choreography. That is, that the reused peers are *conform* [2] to the choreography. Choreographies also support *top-down* development (Fig. 2). Choreographies are first used to obtain local, peer-level, requirements, also called (behavioral) skeletons. Again, one has to be sure that the composition of these peer requirements is conform to the specification, before going on and implementing the peers. The property checking whether a choreography can be realized or not by a set of peers is called *realizability* [3]–[6]. The design of the peer requirements could be achieved *explicitly* by a human as for the choreography. Still, it is much more interesting, following the generative programming paradigm, to retrieve the peer requirements automatically from the choreography specification, in which the peer requirements are *implicitly* defined. This can be achieved using *projection* operations [5], [6], and thus yields a specific notion of realizability targeted at automated choreography-based development: a choreography is realizable if the set of peers that are obtained from it using projection conforms to the choreography itself. If so, a developer may implement the peer requirements by adding business code to them, following the same kind of process than, *e.g.*, when complementing Java RMI skeletons generated from purely functional interfaces. The developer may also use the peer requirements to perform implementation by reuse, using behavioral discovery approaches [7], or even by adapting services that would not perfectly match the needs [8]. If the choreography is not realizable, the designer has to change the choreography, unless corrective solutions are proposed.

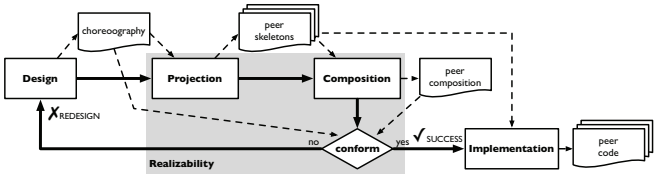


Fig. 2. Top-Down Development Process

Figure 3 presents a simple example of a conversation protocol [3], one of the notations existing for describing choreographies, where each transition is labeled with a message exchanged between two peers, one sender and one receiver.

This protocol involves three peers: a client (cl), a Web application (appli), and a database (db). The client first submits a storage request to the application (req), then the application interacts with a database to store the information (store), and

M. Güdemann is with Systerel, Aix-en-Provence, France.

P. Poizat is with Université Paris Ouest, Nanterre, France and with Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, Paris, France.

G. Salaün is with University of Grenoble Alpes, Inria, LIG, CNRS, France.

L. Ye is with CentraleSupélec, Gif sur Yvette, France and with LRI UMR 8623, Univ Paris-Sud 11, France.

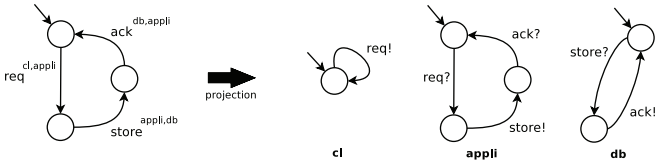


Fig. 3. Example of Conversation Protocol

the database sends back an acknowledgment (ack). On the right hand side of Figure 3, we give the projection obtained from this choreography, where interactions are replaced with message receptions (question marks) and message emissions (exclamation marks). Even with this introductory example, it is difficult to say whether this choreography is realizable. This conversation protocol is actually realizable with synchronous communication but not realizable if peers interact via FIFO buffers (asynchronous communication), because the peer client can send in sequence several req messages whereas in the original contract, these messages can be sent only at a certain moment in time (but for the first one, after an ack interaction).

Motivations. The existing techniques for formally verifying choreographies suffer several drawbacks. First, they are *language-specific*, *i.e.*, they focus on a single choreography modelling language, *e.g.*, UML collaboration diagrams [6], [9], conversation protocols [3], Singularity channels [10] or BPMN 2.0 choreographies [11]. As a consequence, the other existing languages cannot take profit of these analysis techniques, which limits their applicability and impact. Second, most existing works, *e.g.*, [4], [12]–[16], propose techniques for *checking the realizability property only*. Other formal properties and composition issues also need to be verified for choreographies but are hardly tackled in the literature. For instance, if a choreography is not realizable, it is of prime importance from a user perspective to provide *automated solutions for resolving these issues*, that is *enforcing realizability by correcting the possible message sequences*. Third, most existing choreography analysis techniques oversimplify verification by assuming a *synchronous communication model*. This is sufficient in some cases, but the asynchronous communication semantics, used in most distributed systems, raises important theoretical issues such as state space explosion due to communication buffers. These issues are not dealt with by purely synchronous approaches. It should be noted that the use of data being exchanged between peers may also result in state space explosion. Still, this has been recently addressed using symbolic models [17]. Last but not least, but for some exceptions [17]–[19], *limited effort has been spent to develop available formal verification tools* for supporting the choreography-based design of communicating software.

Models. As classified by [20], there are two interaction models for choreography: *interconnected interface models*, where conversations are defined at (each) peer level and interactions are defined by connecting these conversations, and *interaction models*, where interactions between peers are the basic building blocks. The former nicely suits low-level languages such as WS-BPEL where an orchestration would be defined for each peer and communications would

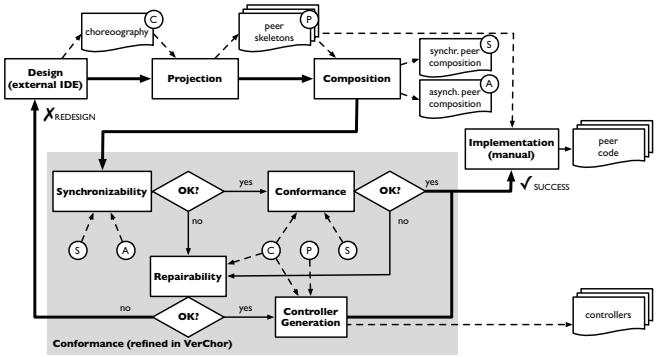


Fig. 4. Top-Down Development Process (Refined) in VerChor

correspond to connections between peer models. However, from a designer perspective, and following the separation of concerns principle, interaction models better suit the needs of choreography specification due to their global perspective. Hence, we focus on interaction-based languages in this article.

Approach. In this paper, we present VerChor, a formal and tool-supported framework available at [21], that supports both bottom-up and top-down development. It also considers both synchronous and asynchronous communication models. As far as verification is concerned, in this article we focus on the compliance between the choreography and its distributed version consisting of interacting peers. This check includes not only the conformance and realizability properties, but also *synchronizability* [15], which addresses conditions under which synchronous and asynchronous peer compositions are equivalent. The benefit of synchronizability is that if a choreography is synchronizable, conformance (hence realizability) can be checked for synchronous communication and yields for asynchronous communication too. Beyond these checks, if the choreography is not realizable, we propose a transparent and non-intrusive solution, which enforces the distributed system to respect the choreography requirements. This is achieved by synthesizing automatically *distributed controllers* that interact together for resolving message ordering issues. However, realizability enforcement is not possible for all choreographies. Therefore, before applying our solution based on controller generation, we need to check whether the *repairability* property is satisfied or not. This check can be achieved on the choreography itself and does not require an analysis of the distributed version of the system.

Figure 4 presents the models being used in a top-down development process, and the order in which properties and synthesis techniques are achieved. This applies also to bottom-up development where peer skeletons are not obtained by projection but are available as an input of the process.

In order to accept the choreography specification languages commonly used by designers as input, we define a *choreography intermediate format* (CIF) and propose automated connections from existing choreography languages (such as conversation protocols or BPMN 2.0 choreographies) to this intermediate format. As far as the back-end connection is concerned, we have developed a translation from our interme-

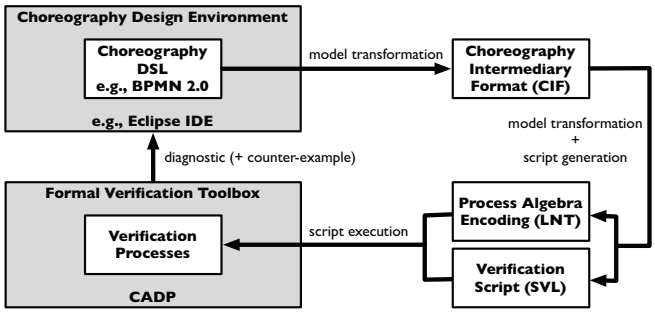


Fig. 5. Framework Overview

diate format to the LOTOS NT (LNT) process algebra [22], which is one of the CADP input specification languages [23]. CADP is a verification toolbox providing a large variety of automated analysis techniques. This connection to LNT/CADP enabled us to develop a fully operational verification library for automating the key choreography synthesis and analysis tasks using behavioral model operations such as products, alphabet hiding, and equivalence checking.

All the steps of our approach, described in Figure 5, are fully automated by model transformations we have implemented, verification scripts we generate, and verification tools we reused from the CADP toolbox.

Other back-end verification toolboxes can be connected to our intermediate format. This requires to develop other translations from the intermediate format to one of the input formal languages of the targeted toolbox. If, for instance, one is interested in using interactive theorem proving for choreographies, an encoding into Isabelle, Coq or PVS input formats could be developed. Translating our intermediate format to Petri net models is also a promising option for reusing existing analysis techniques developed for Petri nets, *e.g.*, [24]–[26].

It is worth noting that, since we have developed a connection to the CADP toolbox through a translation to LNT, all verification tools available in CADP can also be used on the system under design (choreography and distributed implementation). A noticeable example is the Evaluator 4.0 on-the-fly model checker that can verify temporal properties specified in MCL [27], an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators.

Contributions. Our contributions are as follows:

- We propose a *generic, extensible format for describing choreographies* accepting several languages as input (*e.g.*, conversation protocols and BPMN 2.0 choreographies).
- We define a *verification library, automating key choreography analysis tasks* using model and equivalence checking, with a focus on asynchronous communication (via FIFO buffers). Beyond property checking, we also present *controller synthesis techniques* for enforcing realizability.
- We present a set of *freely available tools* we have implemented that supports and automates the different parts of our approach that is (i) translating the choreography description languages accepted as input into the intermediate format, (ii) translating the intermediate format into the input language of the verification tools used for the analysis, and (iii) generating the necessary scripts for automating all the translation and verification steps.

Outline. Section II introduces the choreography languages connected so far to our intermediate format, with a particular emphasis on BPMN 2.0. The choreography intermediate format is itself presented in Section III. In Section IV, we present our verification library. Section V describes the tools we have implemented for supporting our approach. Finally, Section VI reviews related work and Section VII concludes the article.

II. Choreography Description Languages

We have considered the following three groups of interaction-based choreography description languages for designing the intermediate format presented in Section III:

- WSCI and WS-CDL rely on a standard exchange format (XML) which simplifies model Transformation. However, these languages are not systematically equipped with formal semantics and the absence of graphical front-end makes writing painful.
- BPMN 2.0 choreographies [28] or UML collaboration diagrams [9] are user-friendly graphical notations, convenient for end-users, but they often either lack of formal semantics or exhibit various (divergent) ones.
- Chor [5] and conversation protocols [3] are formal description languages equipped with a formal semantics and analysis techniques. Yet, they are difficult for non-experts.

In this section, we introduce conversation protocols and BPMN 2.0 choreographies, because these are the currently available front-ends in the VerChor framework. Yet all the languages mentioned above can be transformed to our intermediate format. We have a particular interest in BPMN 2.0 choreographies, because it became an OMG standard notation in 2011, an ISO standard in 2013 [29], and it is now commonly used for modelling choreographies.

A. Conversation Protocols

A conversation protocol [3] is a Labeled Transition System (LTS) specifying the desired set of interactions from a global point of view. Each transition specifies an interaction between two peers P_{sender}, $P_{receiver}$ on a specific message m. A conversation protocol makes explicit the execution order of interactions. Sequence, choice, and loops are modelled using a sequence of transitions, several transitions going out from the same state, and a cycle in the LTS, respectively.

Definition 1 (Conversation Protocol): A conversation protocol CP for a set of peers $\mathcal{P} = \{P_1, \ldots, P_n\}$ is an LTS $(S_{CP}, s^0_{CP}, L_{CP}, T_{CP})$ where S_{CP} is a finite set of states; $s^0_{CP} \in S_{CP}$ is the initial state; L_{CP} is a set of labels where a label $l \in L_{CP}$ is a tuple m^{P_i,P_j} such that $P_i, P_j \in \mathcal{P}$ are the sending and receiving peers, respectively, $P_i \neq P_j$, and m is a message on which those peers interact; finally, $T_{CP} \subseteq S_{CP} \times L_{CP} \times S_{CP}$ is the transition relation.

A transition $t \in T_{CP}$ is usually denoted as $s \xrightarrow{m^{P_i,P_j}} s'$ where s and s' are source and target states and m^{P_i,P_j} is

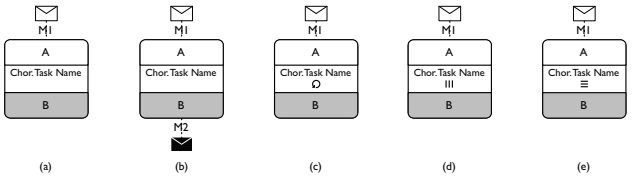


Fig. 6. BPMN 2.0 Notation Primer — Choreography Tasks

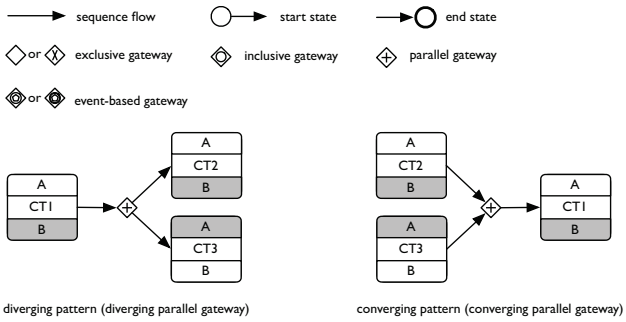


Fig. 7. BPMN 2.0 Notation Primer — Control Flow and Gateways

the transition label. An introductory example of conversation protocol is given in Figure 3.

A conversation protocol is a low-level formal model, which can be computed from other existing specification formalisms such as UML collaboration diagrams [9], Singularity channels [10], or BPMN [11]. It is worth noting that conversation protocols can serve as formal semantic model for the Choreography Intermediate Format (CIF) we present in Section III. However, it is much easier to transform a choreography description language to CIF than going directly to such a low-level model. Indeed, CIF consists of high-level operators that assure a straightforward translation for all afore-mentioned languages, whereas a transformation to conversation protocols requires the flattening of all operators (*e.g.*, expanding a parallel composition to all the possible corresponding interleavings), which is quite difficult, see [11] for a transformation from BPMN choreographies to conversation protocols.

We use LTSs for specifying the peer behavioral model, which defines the order in which the peer messages are executed. A label consists of a message name and a direction (emission ! or reception ?).

Definition 2 (Peer): A peer is an LTS $\mathcal{P} = (S, s^0, \Sigma, T)$ where S is a finite set of states, $s^0 \in S$ is the initial state, $\Sigma = \Sigma^! \cup \Sigma^?$ is a finite alphabet partitioned into a set of send and receive messages, and $T \subseteq S \times \Sigma \times S$ is the transition relation. We write $m!$ for a message $m \in \Sigma^!$ and $m?$ for $m \in \Sigma^?$.

Each peer is obtained by projection from a CP by keeping only messages where that peer appears, and replacing interactions by emissions or receptions.

Definition 3 (Projection): Peer LTSs $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ are obtained by replacing in $CP = (S_{CP}, s_{CP}^0, L_{CP}, T_{CP})$ each label $m^{\mathcal{P}_j,\mathcal{P}_k} \in L_{CP}$ with $m!$ if $j = i$, with $m?$ if $k = i$, and with τ (internal action) otherwise; and finally removing the τ-transitions by minimizing the LTS modulo weak trace equivalence [30], which yields a τ-free and deterministic LTS.

B. BPMN 2.0 Choreographies

BPMN 2.0 [28] (BPMN in the rest of this article) introduces Choreography Diagrams to support conversations with choreography tasks as first class entities. The basic building block of BPMN Choreography Diagrams is a one-way or two-way interaction between peers. This is modelled using a *choreography task* (Figure 6), where interactions involve two peers, A and B, represented by participant bands. A is the initiating peer, *i.e.*, the one that decides when the interaction takes place, it is represented by a white band as opposed to a gray filled band for B. Together with the choreography tasks, there exist message flows relating the interaction with an initiating message (represented by a white envelope) and, possibly, a return message (represented by a black envelope). This yields one-way interactions (Figure 6, (a, c, d, e)) or two-way interactions (Figure 6, (b)). In the rest of this article, for simplification purposes, we assume that message and task names are always identical.

A choreography task may have an internal marker to denote whether, and how the related interaction (one or two message exchanges) is repeated. In a standard loop (Figure 6, (c)), the interaction is performed several times. In multi-instance parallel loops, the interactions are performed by several instances of the choreography task. This can be done in parallel (Figure 6, (d)) or in sequence (Figure 6, (e)). If the exchange is not repeated, no marker is used (Figure 6, (a, b)).

BPMN enables one to describe control flows using sequence flows for performing two tasks in sequence or gateways for more complex behaviors. In our work we take into account the main gateways found in BPMN (Figure 7), that is: exclusive gateways (decision, alternative paths), inclusive gateways (all combinations, from one to all), parallel gateways (creation of parallel flows), and event-based gateways (choice based on events, *i.e.*, message reception or timeout). We require that gateways are either diverging / splitting (multiple outgoing sequence flows and at most one incoming sequence flow) or converging / joining (multiple incoming sequence flows and at most one outgoing sequence flow). Diagrams that would not adhere to this requirement can be transformed by adding new gateways [28], *e.g.*, a gateway being both converging and diverging can be transformed as the sequence of a converging one and a diverging one.

III. Choreography Intermediate Format

In this section, we present the choreography intermediate format (CIF) we propose for automated verification of choreography description languages. Such an intermediate language presents several advantages. First, several input languages can be connected to it, and this allows designers to use their favorite choreography description language. Second, it makes it possible to use jointly several formal verification tools and techniques as back-end, provided that a connection to those tools exists. Third, it can also serve as an expressive standalone specification language for choreographies. Last but not least, the language can be easily extended with new choreography

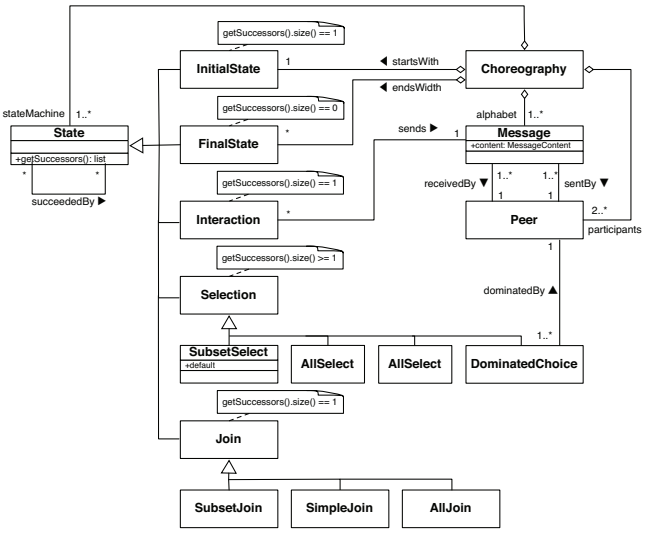


Fig. 8. CIF Meta-Model

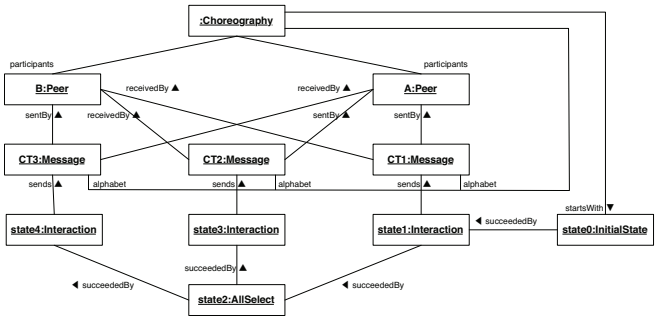


Fig. 9. CIF Model for the Choreography in Fig. 7, left (with an added initial state and message contents abstracted)

constructs, and the framework enriched with other front-end (back-end, resp.) connections from other choreography languages (to other formal verification tools, resp.).

A. A State Machine Meta-model

The CIF meta-model is based on a state machine representation of choreographies, where states model either interactions or choreography operators such as exclusive choice, start of parallel activities, or merging of execution flows. Sequence, *i.e.*, the ordering between states, is modeled using arcs, each arc connecting a source and a target state. Such a meta-model is very close to the meta-models of workflow-based notations, which are the main family of domain specific modeling languages for business processes, choreographies, and orchestrations, with *e.g.*, the WS-BPEL language and the BPMN notation. Indeed, workflows are directed graphs with nodes that correspond not only to gateways, but also to interactions, as demonstrated for example in the BPMN meta-model with ChoreographyTask (the class for interactions), which is a subclass of FlowNode, the class representing nodes in the workflow (these nodes being related by arcs of type SequenceFlow).

A first advantage of such a state machine meta-model is therefore to make it easier to transform workflow-based choreography languages into it, as demonstrated in Figure 9, still without hindering the transformation from other state and transition models such as the conversation protocols presented in Section II. A state machine meta-model also makes it easily possible to represent unbalanced workflows and complex loops (where the flow of execution gets back at some point earlier in the behavior) using arcs between states. Last but not least, the state machine pattern significantly facilitates a further encoding into any formal model of choreographies. This is the case for instance with LNT where some CIF constructs are translated in a straightforward way to the target language (see Section III-C for details), although other constructs deserve more attention.

We give in Figure 8 the CIF meta-model describing more precisely the structure of our intermediate format, expressed as a class diagram. One of the main classes for the state machine representation is the State abstract class which provides an abstract method to access its list of successor states. The InitialState class represents the unique initial state of a choreography. It implements the getSuccessors operation, but has only a single successor state. The FinalState class represents a terminal state of the choreography and will therefore always have an empty list of successors. The Interaction class is used to model a basic choreography interaction through a Message exchanged between a sender and a receiver peer.

The Selection abstract class provides means to split the flow of a choreography into multiple possible continuations. Any instance of a class which extends it will always provide multiple successors through the getSuccessors operation. The most basic variants are Choice and DominatedChoice, where exactly one of the successor states continues. Choice is found in most choreography languages: one branch is executed among a number of possible executions. DominatedChoice is used in some languages (*e.g.*, Chor [5]) for specifying explicitly the peer that makes the choice in case of ambiguity. The parallel execution of all continuations after a selection is represented by the AllSelect class. The parallel execution of a subset of the continuations can be specified using the SubsetSelect class. In this last case, all possible combinations should be possible in the corresponding behavior (*e.g.*, if there are two branches involved we can execute only one of them or the two in parallel) as well as a default branch if such case is specified.

The dual of the Selection class is the abstract Join class, which is realized either by SimpleJoin which has only one incoming active flow, AllJoin for full parallel active flows and SubsetJoin for a subset of all incoming flows. Each join waits for the corresponding number of incoming active flows and synchronizes accordingly. Inconsistencies in a choreography, such as possible mismatches of selection and join operators, can be detected by structural analysis. It does not make sense for instance to match a single choice (Choice) with an all join (AllJoin), as only a single incoming flow is active and should be expected at the corresponding join point.

Figure 9 shows the CIF representation of the choreography in Figure 7, left, with an added initial state. One can easily

identify the participants, represented with objects of class Peer. The exchanged messages are represented by objects of class Message, where each message has an associated sender and receiver. The interactions are represented by objects of class Interaction, with an association to the message which is exchanged, while the parallel gateway is represented by the object of class AllSelect. The sequence in the choreography is represented by the succeededBy association between objects.

One could wonder why we have not chosen other languages as intermediate format such as BPMN, conversation protocols, or the LNT process algebra that we use in the sequel as an intermediate step for generating the behavioral models (LTSs) of peers and choreographies. CIF is close to BPMN in the sense that it consists of high-level operators, but CIF may contain more operators than BPMN, the dominated choice for instance. The main issue with conversation protocols and LNT is that transforming the choreography description languages we consider here (*e.g.*, BPMN) to such languages is quite complex for some constructs (see see Section III-C for details on the encoding from CIF to LNT). One of our main motivations was to make the front-end transformations as simple as possible, which is the case with CIF.

B. Front-end Connections

For illustration purposes, we focus on the subset of BPMN 2.0 choreographies introduced in Section II and show how to transform it into CIF, as illustrated in Figure 9. BPMN ChoreographyTask is transformed into CIF Interaction. If the task is related to a message via a BPMN MessageFlow (as in Fig. 6), we generate a CIF Message from it. Else, we use the ChoreographyTask name (as in Fig. 9). BPMN two-way interactions are first transformed into a sequence of two one-way interactions. BPMN gateways are transformed into corresponding CIF class instances, *i.e.*, exclusive and event-based splits are transformed into Choice instances, parallel and inclusive splits into AllSelect and SubsetSelect constructs, respectively, exclusive and event-based joins into SimpleJoin, inclusive joins are encoded into SubsetJoin, and parallel joins into AllJoin. The sequencing between choreography nodes, achieved with SequenceFlow instances in BPMN, is transformed in CIF using the succeededBy association.

Similarly, the translation of the choreography description languages mentioned at the beginning of Section II is straightforward except for UML collaboration diagrams. Their encoding is slightly more complicated than for the others due to the use of synchronization points between concurrent threads that cannot be encoded using join operators. Therefore, the simplest solution is first to translate UML collaboration diagrams into a lower level formalism, such as conversation protocols or LTSs as done in [6], and then connect this low-level format to our intermediate format. This connection is straightforward for conversation protocols, where sequences and loops are implicitly encoded using arcs in the state machine, and non-deterministic branches are translated to Choice states.

The semantics of our intermediate format is formalized by encoding into LNT (see Section III-C), LNT itself having a formal operational semantics defined in terms of LTSs. Even if the semantics of the input choreography languages are not always formally defined (*e.g.*, conversation protocols are equipped with a formal semantics, while the semantics of BPMN is informally defined [29]), we paid a lot of attention when building our framework to preserve their semantics during the successive translations necessary for making their formal verification possible. We will comment on that with more details in Section V.

C. Back-end Connections

Several back-end connections can be proposed from our intermediate format. Possible candidates are for instance input languages of theorem proving tools (such as Coq, Isabelle, or PVS) and Petri net formalisms. In this article, we focus on input languages for model checking tools, because these verification techniques turn out to be adequate for the properties of interest here (see Section IV). In particular, we propose a connection to the LNT process algebra, which is one of the CADP input specification languages [23].

Now we briefly describe the principles for translating CIF to LNT. More details are available in Appendix, in particular for unbalanced choreographies. The reader can also refer to Section V for technical details of the tool support. For each CIF state, we generate an LNT process as follows:

- Initial/Final state: for the initial state n, suppose $n \rightarrowtriangle m$, *i.e.,* m is the next state of n. The process for n calls the process for m. For a final state, its process does nothing but terminates by using the empty statement (**null**).
- Interaction: each interaction is encoded as an LNT action.
- Selection: we focus here on three types of selection.
  1) all select state n: suppose that n has k outgoing branches, *i.e.,* $n \rightarrowtriangle m_i, i \in \{1, ..., k\}$. The LNT process models the parallel execution of all outgoing branches using the LNT parallel operator (**par**). Each branch $m_i, i \in \{1, ..., k\}$ is translated by a call to the LNT process encoding the node m_i. In addition, if there exists a corresponding all join state mp, we need to generate an additional parallel branch to realize the synchronization point among the different branches. To do so, for this join, we create a synchronization action $sync$ at the beginning of the additional branch and at the end of all other branches. In this way, the additional branch synchronizes with all other branches on $sync$ before calling the process for the next node after mp.
  2) subset select state n: suppose $n \rightarrowtriangle m_i, i \in \{1, ..., k\}$. Any combination of the branches m_i can be executed. For each subset $\{m_{i_1}, ..., m_{i_n}\}, 1 \leq n \leq k, \forall j \in \{1, ..., n\}, i_j \in \{1, ..., k\}$, we obtain all combinations of this subset by using the LNT parallel operator between $m_{i_j}, j \in \{1, ..., n\}$. Then the LNT choice operator (**select**) is used between all combinations of all subsets. If there exists a corresponding subset join state, we generate an additional branch for synchronization purpose as above.

3) choice state n: suppose $n \rightarrow\!\!\!\!\circ\ m_i, i \in \{1, ..., k\}$. The process models the choice execution of all outgoing branches m_i using the LNT choice operator. Each branch calls the LNT process encoding the corresponding node.

- Join: We have three types of join state. A join state has only one outgoing branch.
  1) all join (subset join, resp.) state n: suppose $n \rightarrow\!\!\!\!\circ\ m$. The process for n first synchronizes with all corresponding incoming active branches on the synchronization action $sync$ before calling the process for m, *i.e.,* it corresponds to the additional parallel branch produced by translating the selection (all/subset select) state.
  2) simple join state n: suppose $n \rightarrow\!\!\!\!\circ\ m$, then the process for n calls the process for m. Recall that there is no synchronization point for this state.

Once a CIF instance C has been translated to LNT, one can obtain the corresponding LTS using classical enumerative exploration techniques, *e.g.*, the LNT compilers of CADP. The LTS generated from this LNT specification corresponds to all possible enactments of C.

D. Extensibility

Although CIF covers a large and important part of the possible modelling artifacts for choreographies, there exist possible extensions to the format. An interesting extension to CIF is adding data to message contents [17]. For example, it could be possible to have gateways where the choice among several flows depends on data exchanged in earlier messages. The currently implemented analysis abstracts from the data and can therefore be regarded as an over-approximation. For some of the formal properties of choreographies described in Section IV, data-dependent choices could be helpful. In particular in the case of realizability, where the distributed system obtained after projection is analyzed, data dependency could provide additional means to coordinate choices between distributed peers, which is not possible without data exchange. Data parameters are supported in the LNT formal language, which we use to encode CIF. However, the inclusion of data parameters may increase drastically the size of the LTS model obtained from the LNT encoding, limiting the usefulness of the approach. On the other hand, regarding the properties presented in Section IV, the results of the current encoding without data-dependent choices, can be analyzed a-posteriori to identify false-negatives. For example, it could be verified afterwards that data-dependent choices restore a property like realizability. In such a case our approach would support the choreography designer by highlighting the potential problems.

IV. Verification Library

We present in this section key properties which are of utmost importance when designing choreography-based communicating systems. Following the process described in Figure 4, we proceed as follows. From a choreography model, *projection* is used to retrieve one model for each peer in the choreography. The *synchronous* and *asynchronous compositions* are computed from these behavioral models. *Synchronizability checking* is then achieved by checking the equivalence between these synchronous and asynchronous compositions. If the choreography is synchronizable, *realizability checking* is run, by checking the equivalence between the synchronous composition of the peers and the choreography model. If realizability yields, then the choreography design is fine. If either synchronizability or realizability is not achieved, we run the *repairability* check on the choreography model. If this fails too, the choreography design is incorrect and it must be modified using counter-examples produced in the different verification steps. If the choreography is repairable, then we generate distributed controllers to make the peers behave exactly as prescribed in the choreography. All these tasks are fully automated thanks to the encoding of CIF into the LNT process algebra, and the use of the CADP toolbox for model generation and verification.

The notion of realizability (conformance, resp.) we present in this paper is quite strong yet often used in the literature, see, *e.g.*, [4]–[6]. It ensures that the distributed system exactly reproduces the same sequences of messages as those defined in the choreography. This means that whatever composition is used (synchronous or asynchronous), the visible behaviour must remain exactly the same. This is what the synchronizability property checks. Weaker realizability notions could be considered such as those presented in [12].

A. Synchronizability

Synchronizability is used to check if all interaction sequences in the asynchronous system are also possible in the synchronous one, ensuring that the asynchronous version of the system does not exhibit additional behavior, which is not present in the synchronous composition. These compositions must be the same, otherwise they cannot be conform to the choreography since the asynchronous system diverges somehow by introducing new (unexpected) behaviors.

Definition 4 (Synchronizability): A set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is synchronizable when the synchronous composition of these peers $LTS_s = (\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n)$ is equivalent to their asynchronous composition $LTS_a = (\mathcal{P}_1, \mathcal{B}_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, \mathcal{B}_n)$ (both compositions are defined in [15]), that is, $LTS_s \equiv_t LTS_a$, where $\equiv_t$ stands for weak trace equivalence as advocated in [15] and compares synchronizations in the synchronous composition with emissions from peers to peer buffers in the asynchronous composition.

A recent decidability result [15] proposes the following decision procedure for checking synchronizability: The set of peers is first generated by projecting the choreography specification to each peer, ignoring the messages that are not sent or received by that peer. Then, both the system consisting of peers interacting synchronously and the system consisting of peers interacting via 1-bounded FIFO buffers are computed. Finally, equivalence checking is used to decide whether the two systems are equivalent. If this is the case, the choreography is synchronizable, meaning that the behavior of the distributed implementation will remain the same whatever is

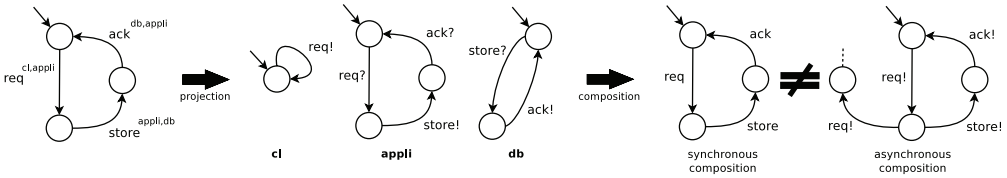


Fig. 10. Example of a Non-Synchronizable Choreography

the size chosen for bounding buffers. This decision procedure relies on bounded, hence finite, systems and thus avoids the generation and analysis of possibly infinite systems.

Definition 5 (Synchronizability Decision): A set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is synchronizable iff $LTS_s \equiv_t LTS_a^1$. In other words: $LTS_s \equiv_t LTS_a^1 \Leftrightarrow LTS_s \equiv_t LTS_a$.

When computing synchronizability, only send actions are considered in the asynchronous case. Ignoring receive actions makes sense for checking synchronizability because: (i) send actions are the actions that transfer messages to the network and are therefore observable, (ii) receive actions correspond to local consumptions by peers from their buffers and can therefore be considered to be local and private information.

We show in Figure 10, the peers obtained by projection from the choreography in Figure 3. This system is not synchronizable, because cl can send several requests (req) in sequence in the asynchronous system, whereas the three interactions req, store, and ack always occur one after the other in the synchronous system, as specified in the choreography.

B. Realizability

This property is used to check if the distributed version of the system behaves exactly as specified in the choreography. This is crucial in a top-down development process in order to ensure that the implementation obtained via projection respects the global specification. Strong notions of realizability can be checked using equivalence checking. Other notions of realizability [12] can be verified similarly, using pre-order simulation or partial order techniques.

Realizability as presented in [15] is checked as follows: one first checks that a set of peers obtained via projection from the choreography is synchronizable. If the synchronizability check returns false, the system is not realizable. Second, if synchronizability is satisfied, the peer composition is computed from the choreography specification: the synchronous version is enough because we know it is equivalent, by synchronizability, to the asynchronous system. We finally compare the choreography with the peer composition, and if they are equivalent, the choreography is realizable.

Definition 6 (Realizability): A conversation protocol C and the set of peers $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ obtained by projection from this choreography (see, Def. 3) are realizable iff the set of peers is synchronizable (Def. 5) and the choreography is equivalent to the synchronous composition, that is, $C \equiv_t LTS_s$.

We recall that a conversation protocol is a low-level formal model, which can be computed from other existing choreography description languages, see Sections II-A and V.

Figure 11 gives an excerpt of a choreography originally presented in [11], where a client (cl) pays a bank (bk), and in sequence, a booking system (bs) stores some information in a database (db) to keep track of a completed transaction. This choreography is synchronizable but not realizable: the resulting (synchronous and 1-bounded asynchronous) compositions are the same, but they are not equivalent to the choreography.

C. Conformance

In a bottom-up development process, peers are being reused and integrated into a new composition. The choreography serves as a contract that the implementation under construction must respect. From a verification point of view, it can be checked exactly as realizability, except that projection is not necessary. Conformance checking takes as input a choreography and a set of peers, whereas realizability checking only requires a choreography specification.

Definition 7 (Conformance): A conversation protocol C and a set of arbitrary peers $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ are conformant iff the set of peers is synchronizable (Def. 5) and the choreography is equivalent to the synchronous composition, that is, $C \equiv_t LTS_s$.

D. Repairability

When a choreography is not realizable, an automated and non-intrusive solution for enforcing realizability is to generate distributed controllers that are in charge of correcting ordering issues to make the corresponding distributed implementation respect the choreography requirements. Repairable choreographies are those for which this controller synthesis solution

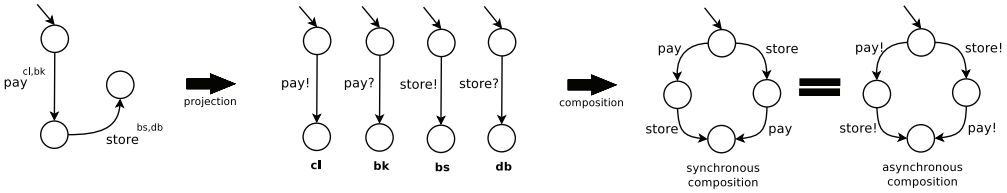


Fig. 11. Example of a Synchronizable but Non-Realizable Choreography

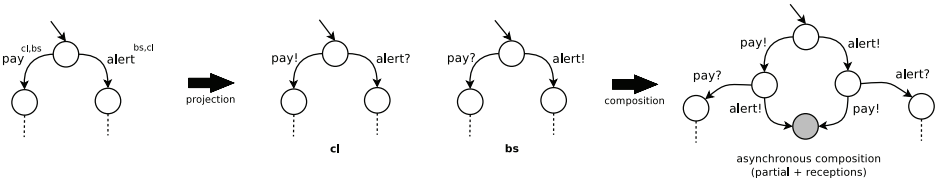


Fig. 12. Example of a Non-Repairable Choreography

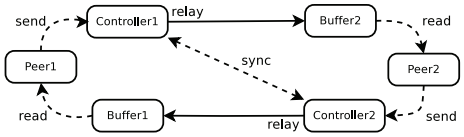


Fig. 13. Architectural View of the System

is possible. But not all choreographies are repairable: A choreography is not repairable when at some point in its behavior there is a choice between interactions involving different sending peers [10]. In that case, realizability cannot be enforced because there is no way to impose the same choice on several distributed peers interacting using asynchronous communication. Therefore, it is important to check whether an unrealizable choreography respects this property before applying the controller generation solution.

Definition 8 (Repairability): A conversation protocol $C = (S, s^0, L, T)$ is repairable if $\nexists\ s \in S$ such that $s \xrightarrow{m^{P_i,P_j}} s', s \xrightarrow{m^{P_k,P_l}} s'' \in T$ and $P_i \neq P_k$.

We can imagine finer notions of repairability, because there are situations where such a *divergent* choice actually corresponds to the start of interleaved behaviors (*i.e.*, this is not a real choice but all possible interleavings of a same set of interactions), and in that case, the choreography is repairable.

Figure 12 presents a partial choreography involving two peers, client (cl) and booking system (bs), which communicate on two messages pay and alert. Here, there is a divergent choice because each peer can take a different decision than its partner, possibly resulting in a deadlock in the system if both peers choose to send, pay (cl) and alert (bs), respectively (grey state in the resulting asynchronous composition, Figure 12, right). Typically, such a situation is not repairable.

E. Control for Enforcing Realizability

If a choreography is not realizable yet repairable, we propose an approach to enforce that the distributed system respects the (synchronizability and) realizability of a choreography by generating distributed controllers [31]. These controllers act locally by interacting with their peer and the rest of the system in order to make the peers respect the choreography requirements. A controller catches local peer emissions and relays them to other peers. Synchronization messages between controllers make them respect the choreography ordering constraints. Figure 13 gives an architectural view of how peers, buffers, and controllers interact altogether.

Definition 9 (Controller): A peer controller is an LTS $\mathcal{C} = (\overline{S}, \overline{s^0}, \overline{\Sigma}, \overline{T})$ where $\overline{S}$ is a finite set of states, $\overline{s^0} \in \overline{S}$ is the initial state, $\overline{\Sigma} = \overline{\Sigma^!} \cup \overline{\Sigma^?} \cup \overline{\Sigma^s}$ is a finite alphabet partitioned into send, locally receive, and synchronization messages. $\overline{T} \subseteq \overline{S} \times \overline{\Sigma} \times \overline{S}$ is the transition relation.

These controllers are obtained by first generating the set of distributed peers by projection from the choreography specification. Then, we check in sequence the system synchronizability and realizability using equivalence checking. If one of these properties is violated, we exploit the generated counterexample to augment the controllers with a new synchronization message. This process is iterated to obtain the controllers via automatic refinement until satisfying both synchronizability and realizability.

A communicating system is controlled if we can synthesize a set of controllers that are able to enforce the peers to realize the choreography specification.

Definition 10 (Controlled System): A set of peers $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ obtained by projection from a conversation protocol C is controlled if there exists a set of controllers $\mathcal{C}_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma_i}, \overline{T_i})$ such that:

- the controlled synchronous composition is equivalent to the controlled asynchronous composition (both compositions are defined in [31]), *i.e.*, $((\mathcal{P}_1, \mathcal{C}_1) \mid \ldots \mid (\mathcal{P}_n, \mathcal{C}_n)) \equiv_t ((\mathcal{P}_1, \mathcal{C}_1, \mathcal{B}_1) \parallel \ldots \parallel (\mathcal{P}_n, \mathcal{C}_n, \mathcal{B}_n))$
- and the choreography is equivalent to the controlled synchronous composition, *i.e.*, $C \equiv_t ((\mathcal{P}_1, \mathcal{C}_1) \mid \ldots \mid (\mathcal{P}_n, \mathcal{C}_n))$

where local interactions (peers to controllers) and interactions between controllers are achieved synchronously, and remote interactions (controllers to peers) are achieved using handshake communication in the synchronous composition and via FIFO buffers ($\mathcal{B}_i$) in the asynchronous composition. In that case, $\equiv_t$ also ignores local interactions from peers to controllers and synchronizations among controllers.

Figure 14 shows the example introduced in Figure 11. While this choreography is not realizable, it is repairable because it does not involve any divergent choice. Non-realizability is caused by peer bs that can send store before peer cl sends pay, and this violates the message ordering as defined in the choreography. We show how controllers for peers cl and bs can solve this problem. Both controllers catch messages sent by their peers. The client controller can immediately forward the payment message to the bank peer. In contrast, the booking system controller is waiting for a message from the client controller (sync_cl_bs) indicating that it can proceed with the emission of the store message. This additional synchronization between both controllers enforces the peers to realize the choreography as shown in the resulting composition (Figure 14, bottom left) where we can see that store! always

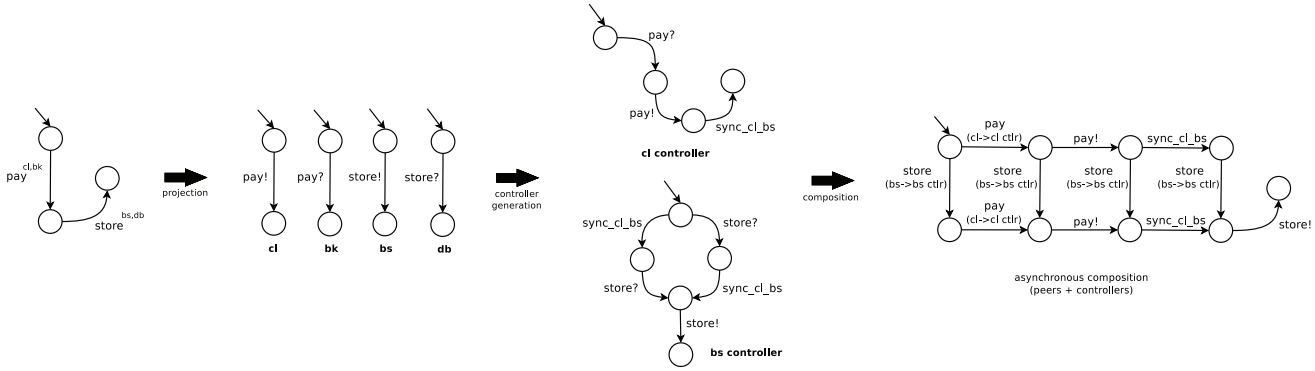


Fig. 14. Example of Repairable Choreographies and Generated Controllers

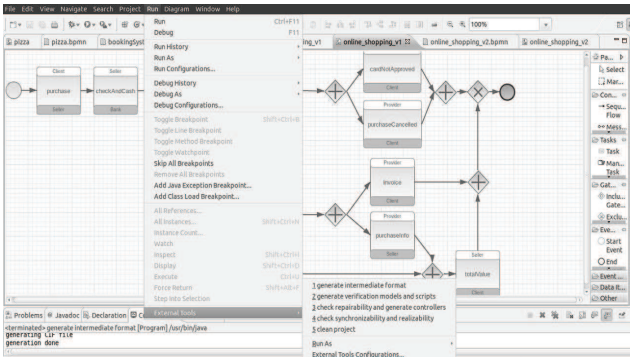


Fig. 15. Connection of VerChor to the Eclipse IDE

appears after pay!.

V. Tool Support and Experiments

A. Tool Support

In this section, we present the tool support [21] that enables the use of VerChor for a fully-automated choreography-based design approach. As presented in Figure 5, VerChor can be applied to a given choreography specification language provided that a model transformation from it to the choreography intermediate language (CIF) is defined. We have chosen to illustrate here the use of VerChor on the BPMN 2.0 standard. In that case the choreography designer can design BPMN choreographies with the Eclipse IDE using the Eclipse BPMN modeler plugin[1]. This plugin is based on an EMF meta-model that is compatible with the OMG BPMN 2.0 specification [28].

We use XML for the textual representation of the intermediate format. Accordingly, we have defined an XML schema (XSD) for it. This schema can be used to validate the (syntactic) correctness of XML intermediate format descriptions. Further, the XML schema can be used to automatically retrieve class implementations of the intermediate format concepts, together with parsers (retrieving object instances from an XML file) and printers (generating an XML file from object instances), *e.g.*, in Java with the JAXB framework.

We have extended the Eclipse IDE, Figure 15, in order to seamlessly integrate our formal verification techniques within the choreography design activity. This is achieved by using the Eclipse IDE external tool extension mechanism. Each tool supporting a formal activity in the choreography design can be called using the Eclipse IDE external tools menu. These tools operate on a given BPMN choreography specification, selected by the designer:

1) *clean project* removes all intermediate files that have been generated for the BPMN specification verification,
2) *generate intermediate format* generates the CIF representation of the BPMN specification (XML file),
3) *generate verification models and scripts* generates the LNT models and the verification scripts (written in the SVL language [32]) from the CIF representation,
4) *check synchronizability and realizability* checks if the BPMN specification is synchronizable and realizable,
5) *check repairability and generate controllers* checks if the BPMN choreography is repairable, and, if so, generates a set of controllers.

We made the choice to let the designer decide in which order to apply the verifications that we propose. Still, one has to apply steps (2) and (3) first to have any of the subsequent verifications working.

The model transformation from BPMN 2.0 into our intermediate format could have been defined using different techniques, *e.g.*, using an XSLT transformation or dedicated model transformation description languages such as ATL. However, to promote modularity and reuse, we have defined it directly in Java. The EMF resource corresponding to the BPMN model within the Eclipse IDE (this is an object instance) is retrieved and analyzed to generate an object instance of the CIF Java meta-model obtained using JAXB (see above). This object instance may then easily be serialized into the CIF XML format using the JAXB-generated XML printers.

As for back-end verification techniques, we have connected our intermediate format to the CADP verification toolbox [23], used here for checking the properties presented in Section IV, with a translation library we implemented in Python. First, we use the PyXB Python library for parsing XML files written using our intermediate format, and for encoding them into a corresponding Python model, which implements classes presented in the meta-model given in Figure 8. Second, we have developed a translation from this Python model for

[1]http://www.eclipse.org/bpmn2-modeler/

TABLE I
EXPERIMENTAL RESULTS

Ex.	Lang.	|P|	|Inter.|	|Sel.|	$\|S\|/\|T\|$	Async. parallel compo. $\|S\|/\|T\|$	Time C | S_c | R | R_p	Results S_c | R | R_p
1	CIF	3	10	1	21 / 29	127 / 200	13s | 1s | 1s | —	√ | √ | —
2	BPMN	6	19	1	580 / 1,828	4,054 / 12,814	86s | 1s | 2s | —	√ | √ | —
3	BPMN	6	19	1	18 / 20	750 / 3,298	83s | 1s | 2s | —	√ | √ | —
4	BPMN	6	19	1	580 / 1,842	16,129 / 51,317	87s | 2s | 2s | —	√ | √ | —
5	CP	7	11	1	11 / 11	158,741 / 853,559	213s | 2s | 2s | 1s	× | × | √
6	BPMN	12	25	4	577 / 2,499	$\sim$1*10^6 / $\sim$7*10^6	648s | 3s | 5s | —	√ | √ | —
7	BPMN	15	31	5	65,556 / 573,479	$\sim$2*10^6 / $\sim$18*10^6	4,711s | 3s | 3s | 5s	× | × | √

choreographies to the LNT process algebra (Section III-C).

CADP tools are convenient for verifying automatically all the properties presented in Section IV, because they enable the verification of choreographies using both model and equivalence checking. Verification of the properties is fully automated thanks to verification scripts generated by our Python translator. It is worth observing that the encoding into LNT also enables other kinds of formal analysis with CADP, such as deadlock search, simulation, or checking temporal properties written in MCL using the Evaluator model checker [27].

An intermediate model in Python code was necessary, instead of translating directly XML to LNT, because we also use Python code for automating various tasks, such as the generation of verification scripts or the analysis of counterexample for distributed controller generation.

The successive encodings (source choreography language, CIF, LNT, and LTS models) on which we rely on in this article must preserve the semantics of the original choreography specification language. Since the final model is an LTS, this is feasible for languages such as conversation protocols or Chor, and it can be verified using trace (or strongest if necessary) equivalence [33]. In contrast, this is much more difficult for notations like WS-CDL or BPMN choreographies. Indeed, these notations do not come with a formal semantics. It is even worse because industrial tools often interpret differently existing standards. Business processes defined with BPMN and the resulting LTS models cannot be compared easily because the first one advocates high-level diagrammatic notation whereas LTSs give low-level flattened views of choreographies. Consequently, in order to validate semantics preservation, model transformations involved in the VerChor platform have been validated experimentally: the results obtained during our experiments were always consistent with the expected verification results. Other techniques such as co-simulation techniques or conformance testing could be considered for comparing both description levels.

B. Evaluation

Table I shows experimental results on some examples of our database, which contains about 400 choreographies, many of them are real-world examples found in the literature, *e.g.*, [3], [6], [10], [11], [15], [28], [31], [34], [35]. Experiments have been carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux. It is worth observing that the translation time (from the input languages to CIF and from CIF to LNT) is negligible even for the largest examples. For each experiment, the table gives the specification language used for describing the input choreography and the size of the choreography in terms of number of peers (P), interactions (Inter.), and selection operators (Sel.). Then, we give the size of the corresponding LTS and the size of the largest intermediate state space for generating the asynchronous version of the distributed system (number of states and transitions). In order to reduce the generation time for compiling the LTS for the asynchronous system, we use recent compositional aggregation techniques [36], which heuristically determine the best sequence of successive composition/reduction for minimizing the intermediate state spaces size. The times for generating all LTSs (C), *i.e.*, synchronous and asynchronous versions of the distributed system, verifying synchronizability (S_c) and realizability (R), and checking whether the choreography is repairable or not (R_p), are given. We have not checked conformance directly when making these experiments because we followed a top-down design approach and used a choreography as input in our experiments. However, the equivalence checking that is central to conformance is used in realizability checking when it comes to compare the choreography model and the product of the peers that have been generated by projection. Finally, the last column details the results for checking synchronizability, realizability, and repairability. Repairability does not need to be checked when the choreography is both synchronizable and realizable.

First of all, when a choreography specification (written in CIF or BPMN for instance) involves parallel operators (AllSelect, SubSetSelect), they are expanded in all the possible interleaved behaviors when the corresponding LTS is generated. This can result in large LTSs (see example 7). We note that the overall time for generating LTSs for choreography and both distributed systems (synchronous and asynchronous) as well as for verifying properties S_c and R is reasonable for medium-size choreographies, see for instance examples 2, 3, 4, 6 in Table I. In any case, even when it takes some time, this is not an issue since these checks are achieved at design-time. In most cases it is more costly to check realizable examples because it deserves an exhaustive exploration of all cases, whereas when the choreography is not realizable, the analysis stops as soon as a violation is found, which can appear early during equivalence computation. We observe that the main cause of explosion, particularly in the asynchronous distributed system and its corresponding computation time, is an increase in the parallelism degree that can arise from (i) the number of peers (*e.g.*, 15 peers in example 7) or (ii) the number of

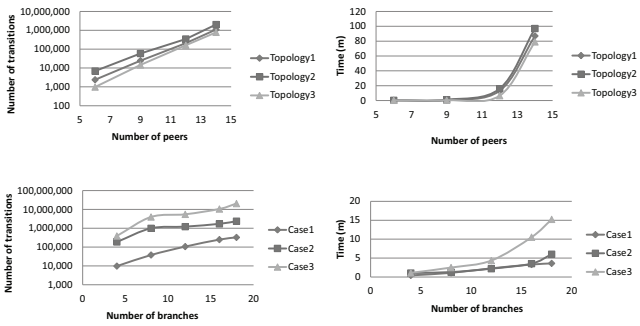


Fig. 16. Experimental Measures: Increased Number of Peers (top) and Increased Number of Branches (bottom)

(interactions in) parallel branches.

In Figure 16, the top part shows that for each topology (choreography with the same structure) the augmentation of peer number leads to an exponential grow in terms of state space size and generation time. Note that we give the largest intermediate number of transitions, which is always larger than the number of states, and represent it using logarithmic coordinates. Similarly, the bottom part of Figure 16 shows that for each case (choreography with the same number of peers and interactions) if we augment the number of parallel branches, size and time gradually increase but less quickly than for the number of peers.

If the choreography is not realizable but repairable, we generate local controllers which synchronize together in order to force the distributed system to respect the order of messages as specified in the global contract. For instance, example 5 presents several ordering issues if peers are generated using projection. In that case, our process requires 6 iterations to construct these controllers, meaning that 6 additional synchronization messages are necessary to make the system realizable. It takes about 20 minutes for this example to successively check synchronizability/realizability using equivalence checking and exploit the resulting counterexample to refine controllers, until completion of the process.

VI. Related Work

The advent of choreography description languages for designing interaction-based systems has raised many issues, such as analysis and verification (projection, realizability, conformance, repair), discovery, code generation, and testing. In this state-of-the-art overview, we will focus on existing works for analyzing and verifying choreography specifications.

Realizability by construction. The results presented in [5], [37], [38] formalize well-formedness rules to enforce the specification to be realizable. More precisely, in [37], [38], Carbone *et al.* identify three principles for global description under which they define a sound and complete end-point projection that is the generation of distributed processes from the choreography. Qiu *et al.* [5] propose a choreography language with new constructs (named dominated choice and dominated loop) ensuring realizability by design. During the projection of these new operators, communications are added to make peers respect the choreography specification. These solutions make the system design more complicated by obliging the designer to specify extra-constraints in the choreography, *e.g.*, by associating dominant roles to certain peers. In [26], Decker and Weske propose a Petri Net-based formalism to specify choreographies. They also define realizability and local enforceability and propose algorithms to check them. However, they consider synchronous communication, and have not investigated mappings from higher-level modeling languages (*e.g.*, UML collaboration diagrams or BPMN).

Asynchronous communication. Several works focused on the realizability problem assuming asynchronous communication. Fu *et al.* [3] proposed three sufficient conditions (lossless join, synchronous compatible, autonomous) that guarantee a realizable conversation protocol. More recently, Basu and Bultan proposed to check choreography conformance and realizability verifying the synchronizability property [15]. Synchronizability compares both the synchronous version of the system with the asynchronous one, and relies on existing finite state verification techniques. [14] studies several notions of realizability and investigates decidability results for choreographies involving services interacting via buffers, which do not assume that messages arrive in the same order in which they have been sent. In [2], the authors tackle the choreography conformance issue from a theoretical point of view, and propose notions of contract refinement and choreography conformance for services that communicate through message queues. [16] proposes techniques to check whether a set of peers interacting asynchronously can realize a choreography with finite buffers, and if so, for what buffer sizes.

Bultan and Fu [9] defined sufficient conditions to check the realizability of choreographies specified with UML collaboration diagrams (CDs). In [6], Salaün and Bultan refine and extend this work with techniques to enforce realizability by adding additional synchronization messages among peers, and a tool-supported approach to automatically check the realizability of CDs for bounded asynchronous communication. The realizability problem for Message Sequence Charts (MSCs) has also been studied (*e.g.*, [4], [39], [40]). [4] for instance presents some decidability results on bounded MSC graphs, which are graphs obtained from MSCs using bounded buffers. These notations are limited because branching and cyclic behaviors are not well supported by CDs and MSCs (*e.g.*, no choice operator and repetition limited to a message at a time in CDs). [41] analyzes the computational complexity of the composition problem, which aims at generating a composition of services interacting via bounded buffers that satisfies a given goal. Our synthesis techniques are quite different because peers are obtained via projection from a choreography specification and controllers non-intrusively monitor those peers to make them respect the choreography ordering constraints.

Realizability enforcement. Lanese *et al.* [42], [43] present a transformation procedure for amending choreographies that does not respect common syntactic conditions for projection correctness. Their approach adds interactions on private operations that make the choreography respect the desired

conditions, while preserving the observational semantics. To do so, they define three connectedness properties (sequence, choice, repeated operation) and show how to enforce each of them, preserving the set of weak traces of the choreography. The main difference compared to our work is that they change the peers' behaviors whereas our approach is non-intrusive and ordering issues are corrected via external controllers.

In [44], the authors present a model-based synthesis process for automatically enforcing choreography realizability. This approach relies on several model transformations for synthesizing the coordination delegates. The provided tool supports the generation of Java code for coordinating, *e.g.*, SOAP-based Web services. This work assumes that actions inside the peers are controllable, which allows to implement an election process in case of divergent choices and pick a winner among the possible senders. Controllability is possible only if developers have preemptively anticipated it. Since this is not always the case, we prefer in our approach to assume that peer actions cannot be controlled. Another work [31] proposes a similar approach for generating distributed controllers enforcing realizability for asynchronously communicating peers. This work tackles this issue from a formal point of view and introduces a sufficient condition for detecting faulty choreographies, that is, choreographies for which realizability cannot be enforced.

BPMN verification. Decker and Weske present, in [45], an extension of BPMN 1.0 (iBPMN) in the direction of interaction modeling. They also propose a formal semantics for iBPMN in terms of interaction Petri nets. Interaction Petri nets are an extension to classical place/transition nets presented in [45] for formalizing choreography semantics through labeling of transitions and thus simplifying the reuse of existing tools for conversation-based languages. At the end of this paper, the authors mention realizability as a novel challenge, but do not give any solution for this issue. Lohmann and Wolf [13] show how realizability can be verified by using existing techniques for the controllability problem, which checks whether a service has compatible partner processes. They mention several models that can be used for modeling choreographies, such as iBPMN, but present their results on multi-peer automata called choreography automata. Their approach works for peers interacting via arbitrary bounded buffers and only consider finite conversations. In [11], the authors have focused on the translation of a subset of BPMN into process algebra for automating the formal analysis of choreographies using model and equivalence checking.

All this related work focuses on specific languages and verification problems (mainly realizability). Our goal is to provide a generic framework, which considers several choreography description languages as input and provides verification primitives for checking some crucial properties in choreography-based design of distributed software, in a fully automated way.

Preliminary versions of this work have been published in [11], [46] and are extended here as follows: we present the Choreography Intermediate Format (Section III), which allows external developers to plug their own languages and tools as front-end and back-end, respectively; we describe in detail the properties that can be analyzed using our framework (Section IV); we present the different components of our verification platform that automates all the checks presented in this article (Section V); we present an extended discussion comparing our approach with related work (Section VI) ; we introduce a new encoding into LNT, which takes unbalanced split/join operators into account (Section III-C, Appendix).

VII. Concluding Remarks

Designing software applications consisting of communicating entities has been greatly simplified with the advent of choreography description languages. Yet, these languages and development processes raise new issues that deserve to be worked out in order to become mainstream in this area. One central problem concerns the correspondence between the global choreography specification and the distributed version of the system composed of a set of peers interacting asynchronously. Beyond providing automated techniques for verifying model compliance, there is also a need for techniques that enforce peers to respect the requirements specified in the choreography. There has been quite some works on these issues, but most results are hardly reusable because they focus on specific notations and do not provide available tool support.

In this article, we have first proposed an intermediate format for describing choreographies. Several interaction-based notations already existing for choreographies (*e.g.*, conversation protocols or BPMN) have been connected to this intermediate format. We have also presented a verification library, which presents a set of key properties that choreographies must respect for ensuring correctness of the system under development. We show how these properties can be automatically verified in practice using model and equivalence checking techniques, via an encoding into process algebra. We had a particular focus on asynchronous communication semantics, that is, peers involved in the distributed version of the system exchange messages via FIFO buffers. Our approach is fully supported by freely available tools that we have implemented [21]. This work can be seen as a first step for joining forces and mutual effort for developing verification techniques and tools for formally analyzing choreographies.

References

[1] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-Services," in *Proc. of ESEC/FSE'09*, 2009, pp. 141–150.

[2] M. Bravetti and G. Zavattaro, "Contract Compliance and Choreography Conformance in the Presence of Message Queues," in *Proc. of WS-FM'08*, ser. LNCS, 2009, pp. 37–54.

[3] X. Fu, T. Bultan, and J. Su, "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services," *Theoretical Computer Science*, vol. 328, no. 1-2, pp. 19–37, 2004.

[4] R. Alur, K. Etessami, and M. Yannakakis, "Realizability and Verification of MSC Graphs," *Theoretical Computer Science*, vol. 331, no. 1, pp. 97–114, 2005.

[5] Z. Qiu, X. Zhao, C. Cai, and H. Yang, "Towards the Theoretical Foundation of Choreography," in *Proc. of WWW'07*. ACM, 2007, pp. 973–982.

[6] G. Salaün, T. Bultan, and N. Roohi, "Realizability of Choreographies Using Process Algebra Encodings," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 290–304, 2012.

[7] D. Grigori, J. C. Corrales, and M. Bouzeghoub, "Behavioral Matchmaking for Service Retrieval," in *Proc. of ICWS'06*, 2006, pp. 145–152.

[8] R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 755–777, 2012.

[9] T. Bultan and X. Fu, "Specification of Realizable Service Conversations using Collaboration Diagrams," *Service Oriented Computing and Applications*, vol. 2, no. 1, pp. 27–39, 2008.

[10] Z. Stengel and T. Bultan, "Analyzing Singularity Channel Contracts," in *Proc. of ISSTA'09*. ACM, 2009, pp. 13–24.

[11] P. Poizat and G. Salaün, "Checking the Realizability of BPMN 2.0 Choreographies," in *Proc. of SAC'12*. ACM, 2012, pp. 1927–1934.

[12] R. Kazhamiakin and M. Pistore, "Analysis of Realizability Conditions for Web Service Choreographies," in *Proc. of FORTE'06*, ser. LNCS, vol. 4229. Springer, 2006, pp. 61–76.

[13] N. Lohmann and K. Wolf, "Realizability Is Controllability," in *Proc. of WS-FM'09*, ser. LNCS, vol. 6194. Springer, 2010, pp. 110–127.

[14] ——, "Decidability Results for Choreography Realization," in *Proc. of ICSOC'11*, ser. LNCS, vol. 7084. Springer, 2011, pp. 92–107.

[15] S. Basu, T. Bultan, and M. Ouederni, "Deciding Choreography Realizability," in *Proc. of POPL'12*. ACM, 2012, pp. 191–202.

[16] G. Gössler and G. Salaün, "Realizability of Choreographies for Services Interacting Asynchronously," in *Proc. of FACS'11*, ser. LNCS, vol. 7253. Springer, pp. 151–167.

[17] H. N. Nguyen, P. Poizat, and F. Zaïdi, "A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies," in *Proc. of ICSOC'12*, ser. LNCS, vol. 7636. Springer, 2012, pp. 525–532.

[18] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography," in *Proc. of ICSE'06*. ACM, pp. 771–774.

[19] X. Fu, T. Bultan, and J. Su, "WSAT: A Tool for Formal Analysis of Web Services," in *Proc. CAV'04*, ser. LNCS, vol. 3114. Springer, 2004.

[20] G. Decker, O. Kopp, and A. Barros, "An Introduction to Service Choreographies," *Information Technology*, vol. 50, no. 2, pp. 122–127, 2008.

[21] "VerChor Framework." `http://lip6.fr/Pascal.Poizat/VerChor/`.

[22] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding, "Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4)," 2011, INRIA/VASY, 149 pages.

[23] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes," in *Proc. of TACAS'11*, ser. LNCS, vol. 6605. Springer, 2011, pp. 372–387.

[24] A. Martens, "Analyzing Web Service Based Business Processes," in *Proc. of FASE'05*, ser. LNCS, vol. 3442. Springer, 2005, pp. 19–33.

[25] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. M. W. Verbeek, "Choreography Conformance Checking: An Approach based on BPEL and Petri Nets," in *The Role of Business Processes in Service Oriented Architectures*, ser. Dagstuhl Seminar Proceedings, 2006.

[26] G. Decker and M. Weske, "Local Enforceability in Interaction Petri Nets," in *Proc. of BPM'07*, ser. LNCS, vol. 4714. Springer, 2007, pp. 305–319.

[27] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in *Proc. of FM'08*, ser. LNCS, vol. 5014. Springer, 2008, pp. 148–164.

[28] *Business Process Model and Notation (BPMN) – Version 2.0*, OMG, january 2011.

[29] ISO/IEC, "International Standard 19510, Information Technology – Business Process Model and Notation," 2013.

[30] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[31] M. Güdemann, G. Salaün, and M. Ouederni, "Counterexample Guided Synthesis of Monitors for Realizability Enforcement," in *Proc. of ATVA'12*, ser. LNCS, vol. 7561. Springer, 2012, pp. 238–253.

[32] H. Garavel and F. Lang, "SVL: A Scripting Language for Compositional Verification," in *Proc. FORTE'01*. Kluwer, 2001, pp. 377–394.

[33] R. Milner, *Communication and Concurrency*, ser. International Series in Computer Science. Prentice Hall, 1989.

[34] "Singularity Design Note 5 : Channel Contracts. Singularity RDK Documentation (v1.1)," 2004, http://www.codeplex.com/singularity.

[35] P. Wong and J. Gibbons, "Verifying Business Process Compatibility," in *Proc. of QSIC'08*. IEEE Computer Society, 2008, pp. 126–131.

[36] P. Crouzen and F. Lang, "Smart Reduction," in *Proc. of FASE'11*, ser. LNCS, vol. 6603. Springer, 2011, pp. 111–126.

[37] M. Carbone, K. Honda, and N. Yoshida, "Structured Communication-Centred Programming for Web Services," in *Proc. of ESOP'07*, ser. LNCS. Springer, 2007, pp. 2–17.

[38] K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," in *Proc. of POPL'08*. ACM, 2008, pp. 273–284.

[39] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 623–633, 2003.

[40] S. Uchitel, J. Kramer, and J. Magee, "Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 1, pp. 37–85, 2004.

[41] P. Balbiani, F. Cheikh, and G. Feuillade, "Algorithms and Complexity of Automata Synthesis by Asynchronous Orchestration With Applications to Web Services Composition," *Electr. Notes Theor. Comput. Sci.*, vol. 229, no. 3, pp. 3–18, 2009.

[42] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the Gap between Interaction- and Process-Oriented Choreographies," in *Proc. SEFM'08*. IEEE Computer Society, 2008, pp. 323–332.

[43] I. Lanese, F. Montesi, and G. Zavattaro, "Amending Choreographies," in *Proc. of WWV'13*, ser. EPTCS, 2013.

[44] M. Autili, D. D. Ruscio, A. D. Salle, P. Inverardi, and M. Tivoli, "A Model-Based Synthesis Process for Choreography Realizability Enforcement," in *Proc. of FASE'13*, ser. LNCS, vol. 7793. Springer, 2013, pp. 37–52.

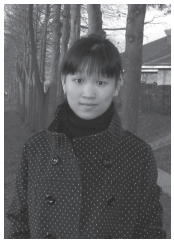
[45] G. Decker and M. Weske, "Interaction-centric Modeling of Process Choreographies," *Information Systems*, vol. 36, no. 2, pp. 292–312, 2011.

[46] M. Güdemann, P. Poizat, G. Salaün, and A. Dumont, "VerChor: A Framework for Verifying Choreographies," in *Proc. of FASE'13*, ser. LNCS, vol. 7793. Springer, 2013, pp. 226–230.

Matthias Güdemann received the PhD degree in Computer Science from the Otto-von-Guericke University Magdeburg, Germany, in 2011. In 2011–2012 he held a post-doctoral position in the research team CONVECS at Inria Rhône-Alpes, focusing on formal verification of BPMN choreography specifications. He is currently a software and systems engineer at Systerel in Aix-en-Provence, France where he is using formal methods for the development of critical systems. His main interest is the application of formal methods to industrial problems.

Pascal Poizat received the PhD degree in Computer Science from the University of Nantes, France, in 2000, and the Habilitation degree in Computer Science from Paris Sud University, France, in 2011. He is currently a full professor at Paris Ouest University and at the LIP6 laboratory (University Pierre et Marie Curie and CNRS). His research activities address software engineering and the use of formal methods in the software development process. This includes supporting the design, verification, adaptation, automatic composition, and testing activities.

Gwen Salaün received the PhD degree in Computer Science from the University of Nantes, France, in 2003, and the Habilitation degree in Computer Science from Grenoble University, France, in 2011. He is currently an associate professor at Ensimag (Grenoble INP) and at the LIG laboratory (University of Grenoble Alpes and CNRS). His research interests include formal methods, automated verification, concurrent systems, software engineering, composition of components and services.



Lina Ye received the PhD degree in Computer Science from University of Paris-Sud 11, France, in 2011. She held a post-doctoral position in the research team CONVECS at Inria Rhône-Alpes, in 2012–2014. She is currently an assistant professor of computing science at CentraleSupélec and at the LRI laboratory (University of Paris-Sud 11), France. Her main research interests include formal methods, model-based design, automated verification of concurrent and distributed systems as well as heterogeneous systems.